**XHR对象**

如果只支持IE7以及更高版本只需要var xhr = new XMLHttpRequest();

如要兼容更早版本或者非主流浏览器var xhr = new XMLHttpRequest() || new ActiveXObject();

1. Xhr用法

首先调用方法open（），传3个参数：请求类型‘get’或‘post’，url，和是否异步（传true则异步）。此方法启动一个请求以备发送，如果是同步的请求，js代码会等待ajax请求响应之后再继续执行。

然后send（）方法传入一个参数：发送的数据，如不发送则传null，因为这个参数对有些浏览器来说是必须的。在收到响应之后，响应的数据会自动填充XHR对象的属性。

Xhr对象有4个属性，responseText，responseXML如果内容类型是‘text/xml’或者’application/xml’，这个属性中保存响应数据的XML DOM文档，status响应的http状态码，statusText http状态码的说明。

接收到响应后第一步检查status属性，一般200算成功，responseText内容 已经就绪，此外304也成功。一般检测到200-300和304状态码算成功。有的浏览器会错误得将204状态码报错。

异步请求的时候还需要判断readyState属性，0：创建对象，1：启动，调用open，2：发送，调用send，3：接收部分，4：接收完成。每次改变值的时候都会触发readyStatechange事件。我们只需要在触发事件时检测readyState值是否等于4以及status的值就能知道请求是否完成。

此外还可以接收到响应之前调用abort（）方法来取消异步请求。如xhr.abort()。调用之后会停止触发事件以及不再允许访问任何属性。终止请求之后还应该对xhr对象进行解引用操作。

1. http头部信息

默认情况下，在发送xhr请求还会发送下列头部信息。

使用setRequestHeather（）方法设置请求头，接受2个参数：头部字段的名称和头部字段的值。且要在open之后，send之前设置

getResponseHeader方法能得到单个响应头部信息，getALLResponseHeaders方法能得到一个包含所有头部信息的长字符串。

1. 默认情况下浏览器对post请求 和web表单的请求不会一视同仁，不过可以使用xhr来模仿表单提交。设置头部‘content-Type’为‘application/x-www-form-urlencoded’,send的数据内容也需要进行序列化，见13章serialize函数。

**XMLHttpRequest 2 级**

1. FormData

现代 Web 应用中频繁使用的一项功能就是表单数据的序列化，XMLHttpRequest 2 级为此定义了FormData 类型。[Dome1](dome1.html)

1. 超时设定

12年只有IE8及以上支持超时设定。

xhr.open("get", "timeout.php", true);

xhr.timeout = 1000; //将超时设置为 1 秒钟（仅适用于 IE8+）

xhr.ontimeout = function(){

alert("Request did not return in a second.");

};

xhr.send(null);

timeout 属性设置为 1000 毫秒，意味着如果请求在 1 秒钟内还没有返回，就会自动终止。**请求终止时**，会调用 ontimeout 事件处理程序。若 readyState同时改变为 4 ，会调用 onreadystatechange 事件处理程序。可是，如果在超时终止请求之后再访问 status 属性，就会导致错误。为避免浏览器报告错误，可以将检查 status 属性的语句封装在一个 try-catch 语句当中。

1. overrideMimeType()方法

用于重写 XHR 响应的 MIME 类型。

当服务器返回的 MIME 类型是 text/plain，但数据中实际包含的是 XML。根据 MIME 类型，即使数据是 XML，responseXML 属性中仍然是 null。通过调用 overrideMimeType()方法，可以保证把响应当作 XML 而非纯文本来处理。这个例子强迫 XHR 对象将响应当作 XML 而非纯文。

var xhr = createXHR();

xhr.open("get", "text.php", true);

xhr.overrideMimeType("text/xml");

xhr.send(null);

**进度事件**

loadstart：在接收到响应数据的第一个字节时触发。

progress：在接收响应期间持续不断地触发。

load：在接收到完整的响应数据时触发。

error：在请求发生错误时触发。

abort：在因为调用 abort()方法而终止连接时触发。

loadend：在通信完成或者触发 error、abort 或 load 事件后触发。

每个请求都从触发 loadstart 事件开始，接下来是一或多个 progress 事件，然后触发 error、abort 或 load 事件中的一个

1. load事件

load 事件，用以替代 readystatechange 事件。响应接收完毕后将触发 load 事件，因此也就没有必要去检查 readyState 属性了。会接收到一个 event 对象， target 属性指向 XHR 对象实例，因而可以访问到 XHR 对象的所有方法和属性。然而，并非所有浏览器都为这个事件实现了适当的事件对象。只要浏览器接收到服务器的响应，不管其状态如何，都会触发 load 事件。而这意味着你必须要检查 status 属性，才能确定数据是否真的已经可用了。Firefox、Opera、Chrome 和 Safari 都支持 load事件。开发人员还是要像下面这样被迫使用 XHR 对象变量因为event不被全部浏览器实现。

var xhr = createXHR();

xhr.onload = function(){

if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304){

alert(xhr.responseText);

} else {

alert("Request was unsuccessful: " + xhr.status);

}

};

xhr.open("get", "altevents.php", true);

xhr.send(null);

1. progress事件

在接受新数据期间，周期性触发。事件对象event包含3个属性：lengthComputable保存进度信息是否可用的布尔值、position已接受字节数、totalSize预期字符数。

xhr.onprogress = function(event){

var divStatus = document.getElementById("status");

if (event.lengthComputable){

divStatus.innerHTML = "Received " + event.position + " of " +

event.totalSize +" bytes";

}

};

为确保正常执行，必须在调用 open()方法之前添加 onprogress 事件处理程序（实际上所以事件最好都在open之前添加）。如果响应头部中包含Content-Length 字段，那么也可以利用此信息来计算从响应中已经接收到的数据的百分比。

**跨资源共享**

CORS（Cross-Origin Resource Sharing，跨源资源共享）是 W3C 的一个工作草案。CORS 背后的基本思想，就是使用自定义的 HTTP 头部让浏览器与服务器进行沟通，从而决定请求或响应是应该成功，还是应该失败。

1. IE对CORS的实现

微软在 IE8 中引入了 XDR（XDomainRequest）类型。能实现安全可靠的跨域通信。XDR 对象的安全机制部分实现了 W3C 的 CORS 规范。以下是 XDR 与 XHR 的一些不同之

处。

cookie 不会随请求发送，也不会随响应返回。

只能设置请求头部信息中的 Content-Type 字段。

不能访问响应头部信息。

只支持 GET 和 POST 请求。

这些变化使 CSRF（Cross-Site Request Forgery，跨站点请求伪造）和 XSS（Cross-Site Scripting，跨站点脚本）的问题得到了缓解。被请求的资源可以设置 Access-Control- Allow-Origin 头部。作为请求的一部分，Origin 头部的值表示请求的来源域，以便远程资源明确地识别 XDR 请求。

XDR 对象的使用方法与 XHR 对象非常相似。也是创建一个 XDomainRequest 的实例，调用 open()方法，再调用 send()方法。但与 XHR 对象的 open()方法不同，XDR 对象的 open()方法只接收两个参数：请求的类型和 URL。

var xdr = new XDomainRequest();

xdr.onload = function(){//请求只要响应有效就会触发 load 事件

alert(xdr.responseText);

};

xdr.open("get", "http://www.somewhere-else.com/page/");//所以的XDR对象都是异步的

xdr.send(null);

在接收到响应后，你只能访问响应的原始文本；没有办法确定响应的状态代码。而且，只要响应有效就会触发 load 事件，如果失败（包括响应中缺少 Access-Control-Allow-Origin 头部）就会触发 error 事件。但没有任何信息可用，就只能确定请求未成功了。

在请求返回前调用 abort()方法可以终止请求：

xdr.abort(); //终止请求

与 XHR 一样，XDR 对象也支持 timeout 属性以及 ontimeout 事件处理程序。

为支持 POST 请求，XDR 对象提供了 contentType 属性，用来表示发送数据的格式，如下面的例子所示。

xdr.open("post", "http://www.somewhere-else.com/page/");

xdr.contentType = "application/x-www-form-urlencoded";

xdr.send("name1=value1&name2=value2");

这个属性是通过 XDR 对象影响头部信息的唯一方式。

1. 其他浏览器对CORS的实现

Firefox 3.5+、Safari 4+、Chrome、iOS 版 Safari 和 Android 平台中的 WebKit 都通过 XMLHttpRequest对象实现了对 CORS 的原生支持。要请求位于另一个域中的资源，使用标准的 XHR 对象并在 open()方法中传入绝对 URL 即可。

与 IE 中的 XDR 对象不同，通过跨域 XHR 对象可以访问 status 和 statusText 属性，而且还支持同步请求。跨域 XHR 对象也有一些限制，但为了安全这些限制是必需的。以下就是这些限制。

不能使用 setRequestHeader()设置自定义头部。

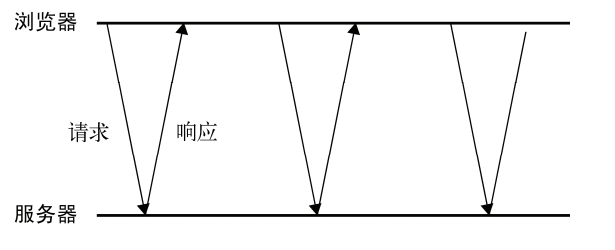
不能发送和接收 cookie。

调用 getAllResponseHeaders()方法总会返回空字符串。

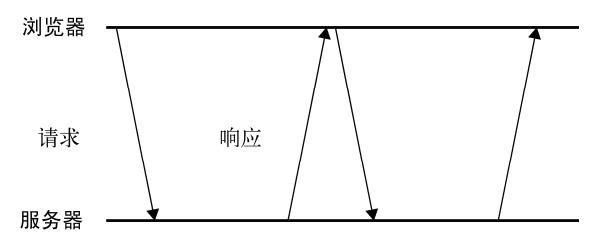
1. Comet

Comet指的是一种更高级的 Ajax 技术（经常也有人称为“服务器推送”）,是一种服务器向页面推送数据的技术。Comet 能够让信息近乎实时地被推送到页面上，非常适合处理体育比赛的分数和股票报价。

有两种实现 Comet 的方式：长轮询和流。长轮询是传统轮询（也称为短轮询）的一个翻版，即浏览器定时向服务器发送请求，看有没有更新的数据。图展示的是短轮询的时间线。



长轮询把短轮询颠倒了一下。页面发起一个到服务器的请求，然后服务器一直保持连接打开，直到有数据可发送。发送完数据之后，浏览器关闭连接，随即又发起一个到服务器的新请求。这一过程在页面打开期间一直持续不断。图展示了长轮询的时间线。



无论是短轮询还是长轮询，浏览器都要在接收数据之前，先发起对服务器的连接。轮询的优势是所有浏览器都支持，因为使用 XHR 对象和 setTimeout()就能实现。而你要做的就是决定什么时候发送请求。

第二种流行的 Comet 实现是 HTTP 流。

为它在页面的整个生命周期内只使用一个 HTTP 连接。具体来说，就是浏览器向服务器发送一个请求，而服务器保持连接打开，然后周期性地向浏览器发送数据。

<?php

$i = 0;

while(true){

//输出一些数据，然后立即刷新输出缓存

echo "Number is $i";

flush();

//等几秒钟

sleep(10);

$i+

在 Firefox、Safari、Opera 和 Chrome 中，通过侦听 readystatechange 事件及检测 readyState的值是否为 3，就可以利用 XHR 对象实现 HTTP 流。在上述这些浏览器中，随着不断从服务器接收数据，readyState 的值会周期性地变为 3。[Dome2](dome2.html)

1. 服务器发送事件

SSE（Server-Sent Events，服务器发送事件）是只读 Comet。SSE API用于创建到服务器的单向连接，服务器通过这个连接可以发送任意数量的数据。服务器响应的 MIME类型必须是 text/event-stream，而且是浏览器中的 JavaScript API 能解析格式输出。SSE 支持短轮询、长轮询和 HTTP 流，而且能在断开连接时自动确定何时重新连接。

**Sse api**

要预订新的事件流，首先要创建一个新的 EventSource 对象，并传进一个入口点：

var source = new EventSource("myevents.php");

注意，传入的 URL 必须与创建对象的页面同源（相同的 URL 模式、域及端口）。EventSource 的实例有一个 readyState 属性，值为 0 表示正连接到服务器，值为 1 表示打开了连接，值为 2 表示关闭了连接。

另外，还有以下三个事件。

open：在建立连接时触发。

message：在从服务器接收到新事件时触发。

error：在无法建立连接时触发。

服务器发回的数据以字符串形式保存在 event.data 中。

source.onmessage = function(event){

var data = event.data;

//处理数据

};

默认情况下，EventSource 对象会保持与服务器的活动连接。如果连接断开，还会重新连接。这就意味着 SSE 适合长轮询和 HTTP 流。如果想强制立即断开连接并且不再重新连接，可以调用 close()方法。

source.close();[demo3](demo3.html)

**事件流**

所谓的服务器事件会通过一个持久的 HTTP 响应发送，这个响应的 MIME 类型为 text/event-stream。响应的格式是纯文本，最简单的情况是每个数据项都带有前缀 data:，例如：

data: foo

data: bar

data: foo

data: bar

对以上响应而言，事件流中的第一个 message 事件返回的 event.data 值为"foo"，第二个message 事件返回的 event.data 值为"bar"，第三个 message 事件返回的 event.data 值为"foo\nbar"（注意中间的换行符）。对于多个连续的以 data:开头的数据行，将作为多段数据解析，每个值之间以一个换行符分隔。只有在包含 data:的数据行后面有空行时，才会触发 message 事件，因此在服务器上生成事件流时不能忘了多添加这一行。

通过 id:前缀可以给特定的事件指定一个关联的 ID，这个 ID 行位于 data:行前面或后面皆可：

data: foo

id: 1

设置了 ID 后，EventSource 对象会跟踪上一次触发的事件。如果连接断开，会向服务器发送一个包含名为 Last-Event-ID 的特殊 HTTP 头部的请求，以便服务器知道下一次该触发哪个事件。在多次连接的事件流中，这种机制可以确保浏览器以正确的顺序收到连接的数据段。

1. Web sockets

要说最令人津津乐道的新浏览器 API，就得数 Web Sockets 了。Web Sockets 的目标是在一个单独的持久连接上提供全双工、双向通信。在创建了 Web Socket 之后，会有一个 HTTP 请求发送。在取得服务器响应后，建立的连接会使用 **HTTP 升级从 HTTP 协议交换为 WebSocket 协议**。也就是说，使用标准的 HTTP 服务器无法实现 Web Sockets，只有支持这种协议的专门服务器才能正常工作。

由于 Web Sockets 使用了自定义的协议，所以 URL 模式也略有不同。**未加密的连接不再是 http://，而是 ws://；加密的连接也不是 https://，而是 wss://**。在使用 Web Socket URL 时，必须带着这个模式，因为将来还有可能支持其他模式。

**使用自定义协议而非 HTTP 协议的好处是，能够在客户端和服务器之间发送非常少量的数据**，而不必担心 HTTP 那样字节级的开销。由于传递的数据包很小，因此 Web Sockets 非常适合移动应用。使用自定义协议的缺点在于，**制定协议的时间比制定JavaScript API 的时间还要长。**Web Sockets 曾几度搁浅，**就因为不断有人发现这个新协议存在一致性和**

**安全性的问题。**目前支持 Web Sockets 的浏览器有 Firefox 6+、Safari 5+、Chrome 和 iOS 4+版 Safari。

Web Sockets API

要创建 Web Socket，先实例一个 WebSocket 对象并传入要连接的 URL：

var socket = new WebSocket("ws://www.example.com/server.php");

注意，必须给 WebSocket 构造函数传入绝对 URL。

var socket = new WebSocket("ws://www.example.com/server.php");

WebSocket表示当前状态的 readyState 属性。readyState的值永远从 0 开始。

WebSocket.OPENING (0)：正在建立连接。

WebSocket.OPEN (1)：已经建立连接。

WebSocket.CLOSING (2)：正在关闭连接。

WebSocket.CLOSE (3)：已经关闭连接。

要关闭 Web Socket 连接，可以在任何时候调用 close()方法。调用了 close()之后，readyState 的值立即变为 2（正在关闭），而在关闭连接后就会变成 3。

socket.close();

发送和接收数据

要向服务器发送数据，使用 send()方法并传入任意字符串，例如：

socket.send("Hello world!");

当服务器向客户端发来消息时，WebSocket 对象就会触发 message 事件。这个 message 事件与其他传递消息的协议类似，也是把返回的数据保存在 event.data 属性中。

socket.onmessage = function(event){

var data = event.data;//字符串

//处理数据

};

其他事件

WebSocket 对象还有其他三个事件，在连接生命周期的不同阶段触发。

open：在成功建立连接时触发。

error：在发生错误时触发，连接不能持续。

close：在连接关闭时触发。

在这三个事件中，只有 close 事件的 event 对象有额外的信息。这个事件的事件对象有三个额外的属性：wasClean、code 和 reason。其中，wasClean 是一个布尔值，表示连接是否已经明确地关闭；code 是服务器返回的数值状态码；而 reason 是一个字符串，包含服务器发回的消息。可以把这些信息显示给用户，也可以记录到日志中以便将来分析。

1. SSW与Web Sockets

面对某个具体的用例，在考虑是使用 SSE 还是使用 Web Sockets 时，可以考虑如下几个因素。**首先，你是否有自由度建立和维护 Web Sockets 服务器？**因为 Web Socket 协议不同于 HTTP，所以现有服务器不能用于 Web Socket 通信。SSE 倒是通过常规 HTTP 通信，因此现有服务器就可以满足需求。

**第二个要考虑的问题是到底需不需要双向通信**。如果用例只需读取服务器数据（如比赛成绩），那么 SSE 比较容易实现。如果用例必须双向通信（如聊天室），那么 Web Sockets 显然更好。别忘了，**在不能选择 Web Sockets 的情况下，组合 XHR 和 SSE 也是能实现双向通信的。**

**安全**